Choice and Performance in Locking for Databases

Y.C. Tay
Harvard University and National University of Singapore

Rajan Suri
Harvard University

1. Introduction

Locking is the most popular database concurrency control algorithm. There is a considerable amount of flexibility in the implementation of locking, and some of the choices entail significant differences in performance. This paper addresses three of the most important choices -- the choice of granularity of locks, the choice of conflict-resolution technique, and the choice of when to set locks -- and their performance implications. We will describe these three choices in turn.

The granularity of a lock refers to how much data the lock locks; it is coarse if each lock covers a large amount of data, and fine otherwise. If the granularity is coarse, few transactions can run concurrently. If it is fine, a transaction will have to set many locks, thus incurring more locking overhead, and also increasing the chances of conflict. What is the appropriate choice of granularity?

Work supported by US National Science Foundation Grant ECS-82-03925.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Locking, like all concurrency control algorithms, resolves conflicts by a combination of restarts and blocking. It is easy to see why restarts degrade performance -- the computation done before the restart is wasted. Performance degradation due to blocking is more subtle : blocking prevents a transaction from doing anything with the data it locks, even while other transactions are trying to access the data. Which of these two conflict-resolution techniques is preferable?

In practice, a transaction usually acquires locks only when it needs them ; this is called dynamic locking. There is an alternative to dynamic locking, where a transaction acquires all the locks it needs before it starts execution. This way, once a transaction begins computation, termination is assured, since there would be no restarts due to deadlocks. Hence, this locking policy, called static locking or predeclaration, is usually thought of as a strategy for deadlock avoidance [C, DC, G1]. Dynamic locking has the advantage of not locking data before they are needed, but it leads to restarts. On the other hand, static locking locks data before they are needed, but it does not cause restarts. Which is better?

Choice of granularity has been studied before in [R, MK]; the effect of restarts and blocking on the performance of concurrency control algorithms was examined in [BBD] ; and comparisons of static and dynamic locking were made in [C, DC, R]. All these studies are by simulation. We shall use an analytic model instead. In [T], we introduced a model for locking and used it to study several systems. This paper reports the model's conclusions about the three choices :

(1) The appropriate choice of granularity depends on which part of a general granularity curve the system sees.

(2) Conflicts should be resolved by blocking if the workload is light, but restarts may be preferable if the workload is heavy. Furthermore, a pure restart policy, where all conflicts are resolved by restarts, can perform as well as a strategy using both blocking and restarts.

(3) If resource contention is not excessive, then dynamic locking is better than static locking if the workload is light, but static locking is better if the workload is heavy.

Section 2 is a review of the model, and Sections 3, 4, and 5 study separately the three problems. We will only present the results from the model. Details on how these results are obtained can be found in [T]. Along the way, we shall mention how our results relate to those in the literature. Again, we will be brief, and details can be found in [TGS]. We should emphasize that, first, ours is an analytic study of these choice problems, and second, this study of these three very different problems is based on a single model.

2. The Model

We first describe the locking protocol.

The database is a collection of *data granules*. A granule may be a file, a page or a record. Users access the database with *transactions*. Before accessing any data, a transaction must first lock the granule containing that data. If the granule is already locked (there is a *conflict*), the transaction must wait in a queue for the lock to be released (it is *blocked*). If this causes a deadlock, then the transaction is aborted : it releases its locks, waits for some time to let the conflicting transaction terminate, then starts again. Transactions release their locks when they terminate (successfully complete). When a lock is released, the first transaction in the queue for that lock acquires that lock.

We now describe the model.

Let D be the number of granules. We assume that all transactions require the same number of locks, k. A transaction makes a sequence of $k+2$ requests : the 0-th request is a request to start, the $(k+1)$-st is for termination, and the i-th, where $1 \leq i \leq k$, is for a lock on a granule. We assume the inter-request time is uniformly distributed on $(0, 2T)$ (so the average inter-request time is T). A transaction does not request a lock it already holds. The sequence of requests a transaction makes is called its *script*.

All requests are handled by the *scheduler*. Requests to start and terminate are immediately granted. Lock requests are handled according to the locking protocol. Upon termination, a transaction is immediately replaced by another transaction. Similarly, when a transaction T' is aborted (because of a deadlock), another transaction T'' takes its place, while T' waits to be readmitted. The number of executing and blocked transactions (excluding those aborted transactions that are waiting) is therefore a constant, say N. An aborted tansaction retains its script. However, since the transaction is replaced by another transaction, it *looks like* the transaction releases its locks and restarts immediately with a new script.

We assume all locks are *exclusive* , i.e. two transactions may not share the same lock. Furthermore, there is *uniform access* , i.e. the probability that a transaction requests for a particular granule it has not locked is the same as that for any other granule. These assumptions are not restrictive. In [T], we show that systems with shared locks, or with nonuniform access like the 80%-20% rule in [MK, LN], are in fact equivalent to systems with only exclusive locks and uniform access.

Some of the above information is summarized in the flow diagram in Fig. 2.1, which charts the progress of transactions. We refer to each node in the diagram as a *stage*.

If the transactions originate from interactive terminals, one may view the flow diagram as a module in a larger model, as shown in Fig. 2.2. Here, the transactions first wait at a queue before entering the system (which, unless otherwise specified, always refers to the flow diagram). The queue exists if there is a maximum multiprogramming level, and the number of terminals exceeds it. The multiprogramming level N refers to the number of transactions in the executing and waiting stages, and excludes those in the queue. When a transaction terminates, it returns to the user at the terminal, who sends another transaction after some time lapse. Solution of this network consists of two steps : (1) solving our model to determine the throughput for a given N, and (2) solving the network by using the usual flow-equivalent methods from queueing network modeling [CS, D]. We will only be concerned with the first step.

With the help of the flow diagram, we can derive a set of equations describing the behaviour of the system. In deriving these equations, only the steady state average values of the variables are used ; henceforth, we will consistently refer only to steady state average values. The two principal performance measures we are concerned with are the throughput t and restart rate $a=\sum_{j=0}^{k-1} a_j$ (a_j as in Fig. 2.1). In the analysis, we make one major assumption : the restart rate is small compared to the throughput, i.e. $\frac{a}{t} << 1$. (This is true in real systems -- see [BO, G2].)

The performance of locking is governed by two factors : resource contention and data contention. The former refers to contention over memory space, computing time and other resources, and the latter refers to contention over data. Essentially, data contention determines the number of executing transactions $N_e = \sum_{j=0}^{k} N_j$ (N_j as in Fig. 2.1), while resource contention determines the rate of execution of a transaction between its lock requests. (Blocked transactions do not compete for resources, so this rate depends on N_e; it decreases if N_e increases.) The model separates these two forms of contention, so that we may study each in turn, then their interaction (see the next section).

Now, suppose we turn off the concurrency control, so all that the transactions suffer from is resource contention. For high enough loads, the system may thrash, i.e. the throughput first increases, then decreases. This is the usual thrashing behaviour in operating systems [DKLPS]. Since this thrashing is due to resource contention, we call it *RC-thrashing* . Conversely, suppose the processor has enough resources (space, computing power, etc) to make resource contention negligible, so all that the transactions suffer from is data contention. The model shows that, for high enough loads, the system will thrash, and we call this phenomenon *DC-thrashing* . The model also shows that DC-thrashing can occur *even if the restart rate is low* . This implies that blocking can cause thrashing — excessive time loss in queues for locks will lead to a drop in throughput. (This phenomenon was observed by Balter, Berard and Decitre in their simulation studies [BBD].) DC-thrashing occurs when the parameter $\frac{k^2N}{D}$, which we call the *workload*, is about 1.5 . Since a system should not operate in the thrashing region, we have the following :

Rule of Thumb The workload $\frac{k^2N}{D}$ should not exceed 1.5

DC-thrashing thus limits the number (N) of transactions in a system, and the number (k) of locks they may request. We call the region in the parameter space where $\frac{k^2N}{D} < 1.5$ the *operating range* . Workload is said to be *light* in the operating range, and *heavy* outside it. In the operating range, the model gives the following estimates of the throughput and restart rate:

$$t = \frac{N}{(k+1)T}\left(1 - \frac{k^2\lambda}{4.5}\right) \qquad (2.1)$$

$$a = \frac{k^2(k-1)^2\lambda^2(4.5 - 2k\lambda)}{4.5T(9k - k^2\lambda + 4.5 - 2k\lambda)} \qquad (2.2)$$

where $\lambda=\frac{N}{D}$.

This sums up the model and relevant results. The next three sections will discuss the three problems of choice.

3. Fine vs Coarse Granularity

We have defined a data granule to be the unit of locking : a transaction can lock, and can only lock, one granule at a time. However, a granule may contain one or more *data items*, which are the units of data. It is the data item that a transaction wishes to access. If a granule contains 20 items, and a transaction locks a granule to update one of the items, then the transaction prevents others from accessing the other 19 items as well.

Let M (a multiple of D) be the number of items, so there are $\frac{M}{D}$ items per granule. The granularity is decided by the choice of D, and this choice affects the value of k, for the following reason. Suppose a transaction wishes to access l items. Under uniform access, two items may belong to the same granule, so the transaction length k is, in general, less than l. The expected value of k depends on M, D and l, and is given by the following formula [Y] :

$$k = D\left(1 - \prod_{i=1}^{l} \frac{M(1-\frac{1}{D})-i+1}{M-i+1}\right) \qquad (3.1)$$

Graph 3.1 shows how k varies with D for a given l when $M=100$. Note that, initially, k increases proportionately with D, but is eventually insensitive to changes in D.

If we substitute (3.1) into (2.1), then we can determine how changing D (i.e. the granularity) affects t. A change in granularity affects the performance through three factors : (1) locking overhead — an increase in D increases k, and therefore increases the locking overhead ; (2) data contention — changes in D and k change the workload $\frac{k^2N}{D}$, and thus affect the data contention ; and (3) resource contention — a change in data contention in turn changes the number of executing transactions N_e, and hence changes the level of contention over computing resources. These three factors combine to shape the *granularity curve* in Graph 3.2a, which shows the effect of granularity on throughput in the operating range. The curve shows that, initially (before point A in Graph 3.2a), the throughput drops as we refine the granularity. This happens because when D increases, k increases proportionately, thus worsening the workload $\frac{k^2N}{D}$. Excessive data contention caused by the increased workload, and the increasing locking overhead, lead to the drop in throughput. However, with further refinement of granularity, k becomes insensitive to changes in D, and data contention slackens. The increased concurrency overcomes the increasing locking overhead (since k is still increasing), and throughput increases (between points A and B). To understand the decrease in throughput towards the end of the curve, note that data contention alleviates resource contention by blocking some of the transactions. Suppose now that we load the system with enough transactions to cause RC-thrashing if the concurrency control is turned off. Further, suppose the concurrency control, when turned on, will block enough transactions to avoid RC-thrashing. Now if we refine the granularity, then the data contention diminishes because of increased concurrency. Resource contention then dominates, and thrashing results.

Depending on the combination of parameters, all or only part of the granularity curve in Graph 3.2a may be manifest. For example, if the transactions are too long, the effect of the locking overhead may persist for so long that soon after the point A, D reaches the maximum possible value M, i.e. one item per granule (see Graph 3.2b). On the other hand, for short transactions and non-excessive loads, the regions of excessive data and resource contention may be imperceptible or absent (see Graph 3.2c). Judging from the throughput alone, the transactions should be required to lock the whole database (D = 1) in the case of Graph 3.2b. In the case of Graph 3.2c, however, the granularity should be as fine as possible. These conclusions remain valid if we also consider the effect of granularity on the number of restarts per completion (see [T]).

Curves similar to those in Graph 3.2 were observed by Ries and by Munz and Krenz in their simulations [R, MK]. Graph 3.2 also explains the effect of granularity in Carey's simulations [C].

4. Restarts vs Blocking

We call the locking policy we have been considering the *waiting case*, since transactions may wait for locks. By the approximation $\frac{a}{t} << 1$, restarts are rare, so almost all conflicts are resolved by blocking in the waiting case. In contrast, in [T], we consider a locking policy where all conflicts are resolved by restarts : whenever there is a conflict, the requesting transaction is restarted. We call this the *no waiting case*. (Note: The no waiting case does not use the approximation $\frac{a}{t} << 1$.) Since one case uses only restarts, and the other uses blocking almost exclusively, a comparison of the two cases should bring out the difference in effect of restarts and blocking on locking performance.

Theorem [T] Let t_w be the throughput for the waiting case and t_n the throughput for the no waiting case. For the same D, k, N and T, and $2 \leq k \leq 20$, if t_n is less than t_w, then the difference is less than 5%; furthermore, t_n exceeds t_w if the workload is sufficiently heavy.

Graph 4.1 compares the throughputs and restart rates of the two cases for $k=2$, $D=40$, and $T=1$, and Graph 4.1a illustrates the above result. We find this result surprising. Intuitively, a pure restart strategy is so severe that we would expect its performance to be very bad. Yet, the above result says that, in terms of throughput, the no waiting case is as good as, if not better than, the waiting case. However, note from Graph 4.1b that, as expected, the no waiting case has a much higher restart rate.

One may be tempted to conclude that the no waiting case is inferior because its restart rate is high, even if the throughput is good (compared to that of the waiting case). But a high restart rate is not bad *per se*. For transactions that do not communicate with the user, we can make restarts transparent to the user, so that all he can observe are the throughput and response time. Nonetheless, restarts waste resources. If the restart rate is high, the user must become aware of this wastage through observing the throughput and response time alone. Is something missing from our model?

In our transaction model, restarts are instantaneous. However, since restarting a transaction involves releasing its locks, and possibly restoring the values it has changed, restarts may take a significant amount of time. To model this, we add a time delay for each request that a restarting transaction has made (see Fig. 4.1). Let the time spent by a transaction in stage A_i be bT for any i. Hence, in Fig. 2.1, $b = 0$.

Graph 4.2 compares t and a for $b=0$ and $b=0.5$. (This graph also indicates simulation results. Details about the simulation can be found in [T].) Indeed, the time delays do slow down the throughput, and the restart rate as well. However, the decrease in throughput is only about 10% at the waiting case thrashing point, although each time delay is 50% of the inter-request time.

Another factor that can increase the difference in throughput of the waiting and no waiting cases is resource contention. In the result above, we used the same T for the two cases. However, for the waiting case, as much as a third of the N transactions are blocked. The number of executing transactions N_e in the waiting case is therefore less than that in the no waiting case. Since T is in general an increasing function of N_e, T is larger for the no waiting case. Hence if we introduce resource contention in Graph 4.1, the t_n curve will be depressed more than the t_w curve, thus increasing the gap between the two before they cross.

A third differentiating factor depends on the transaction environment. Recall from Section 2 that, after a restarted transaction has released its locks, it is held back for some time while another transaction takes its place. This delay (the *conflict-avoidance delay*) is to avoid conflicting with the transaction that caused it to restart. In a batch environment, it does not matter how many transactions are held back this way. For on-line systems, however, this delay must be taken into account. Fig. 4.2 shows how Fig. 2.2 is changed if conflict-avoidance and restart delays are incorporated.

Let S be the (average) conflict-avoidance delay. To avoid repeating the conflict, S must be long enough to let the conflicting transaction terminate. The time it takes for a transaction to terminate, without restarting, is $(k+1)T$ for the no waiting case, and $\frac{N}{t}$ for the waiting case (since restarts are rare). Hence S must be of the order $\frac{k+1}{2}T$ for the no waiting case, and $\frac{N}{2t}$ for the waiting case.

Graph 4.3 uses these values of S to compare the response times of the two cases for two sets of parameters. Note that the response time for the no waiting case is higher than that for the waiting case before the latter's DC-thrashing point, although the throughputs are similar by the theorem. This is the effect of the conflict-avoidance delays. On the other hand, without these delays, the restarting transactions will have a higher probability of conflict, so that the throughput will be less. Thus, the no waiting case must suffer a tradeoff in throughput and response time, and is in this sense inferior to the waiting case in the operating range.

We now review the theorem in the light of the preceding analysis. The three factors we just considered identify the conditions under which the no waiting case is either better, or not much worse, than the waiting case. These conditions are: (1) low restart cost (Gray observed that restarts in centralized systems are no big deal since the necessary mechanisms are simple and already there for other purposes [G1]), (2) little resource contention --- the inter-request time T should not be too sensitive to the number of executing transactions, and (3) batch processing, where response times are immaterial. Put in another way, one's intuition that restarts are bad is invariably based on a violation of one of these conditions.

The fact that the no waiting case may be better eventually shows that *locking with no waiting is a way to overcome the limitation that blocking imposes on the waiting case.* The choice between restarts and blocking as a conflict-resolution technique thus comes down to this: if the three conditions above are satisfied, then blocking is preferable when the workload is light, but restarts are preferable when the workload is heavy.

If condition (1) or (2) is violated, then the throughput curve for the no waiting case will be depressed more than that for the waiting case, so that the former may always be less than the latter. This explains why Carey concluded from his simulations that blocking is always preferable to restarting [C].

5. Static vs Dynamic Locking

In comparing static and dynamic locking, we shall assume that the lock requirements of a transaction are the same. In practice, however, predeclaration often requires locking more than a transaction will eventually use because of uncertainty over what it might need [G1].

In [T], we studied two forms of static locking -- incremental and atomic. Here, we will restrict ourselves to atomic static locking. In (atomic) static locking, when a transaction starts, it submits to the scheduler the list of locks it needs. The scheduler checks the list to see if all the requested locks are available. If so, it grants those locks, and the transaction begins execution ; otherwise, the transaction has to wait until all the locks it needs are available.

Our model for static locking is as follows: Consider the no waiting case, and let the inter-request time be 0 for stages N_1, N_2, ... , N_{k-1}, and $(k+1)T$ for stage N_k. It follows that when a transaction submits its script, the scheduler will grant locks to the transaction as it scans the script, restarting the transaction as soon as there is a conflict. Since the inter-request time is 0 except for the last stage, if the scheduler can grant all the requested locks, then the effect is to grant them in one atomic step. (This means that any cost in time in the predeclaration will have to be charged to the last stage N_k.) Furthermore, in our model, a restarted transaction suffers a conflict-avoidance delay (see the previous section); hence, the effect is to make a transaction wait some time before trying again whenever its predeclaration encounters a conflict. (Thus the number of 'restarts' a transaction suffers in this model is really the number of tries it makes before getting its locks.)

Comparisons of dynamic and static locking have been done before by Carey, Devor and Carlson, and Ries [C, DC, R]. Their comparisons are based on the number of transactions N' in the expanded system in Fig. 4.2. Graph 5.1 is such a comparison using our analytic model. It is similar to the comparison of the waiting and no waiting cases in that dynamic locking is better for light workloads, whereas static locking is better for heavy workloads. There are similar conclusions in [DC, R]. Note that $T=1$ in the graph, so resource contention has been factored out. As in the previous section, when we include resource contention, the throughput curves will be depressed by different amounts, so that one may always be better than the other. For example, when the workload is light, dynamic locking will have more executing transactions than static locking, so that the effect of resource contention on dynamic locking is greater than that on static locking. Thus, resource contention may depress the throughput curve for dynamic locking in Graph 5.1 to below that for static locking. This is the case in [C], where Carey concluded from his simulations that static locking is better even for light workloads.

The conclusion that static locking is better for heavy workloads is not surprising, since it agrees with one's intuition. What is surprising is that it holds under the assumption that $\frac{a}{t} << 1$ for dynamic locking. Now recall that the motivation for static locking is to avoid deadlocks. Here, we find that, even if deadlocks are rare, it still pays to do static locking if using dynamic locking will lead to excessive blocking (see Section 1). Like dynamic locking with no waiting, (atomic) static locking is a way to overcome the limitation on workload that blocking imposes on dynamic locking.

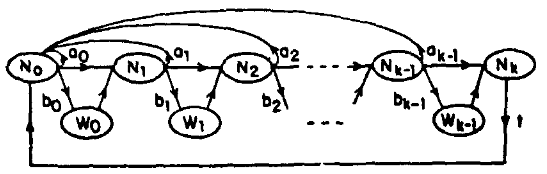
Letting a transaction wait for a lock preserves its results; however, this also allows the transaction to selfishly hold on to locks while not using them. If, on the other hand, a transaction restarts whenever it encounters a conflict, it will waste what it has already done, but will hold locks only for as long as it needs them. This is an altruistic strategy that pays off when the workload is sufficiently heavy: the comparison of the waiting and no waiting cases shows that, under the right conditions, the latter eventually outperforms the former. A pure restart strategy thus offers a way of overcoming the limit on workload that blocking imposes.

When comparing static to dynamic locking, it is often said that dynamic locking has the advantage of holding locks for a shorter period, whereas static locking has the advantage of avoiding deadlocks. This is true when the workload is light, but when it is heavy, the blocking in dynamic locking in fact causes the transactions to hold locks for longer than in static locking. That is why dynamic locking works better for light workloads (if resource contention is not intense), but is inferior for heavy workloads. As for deadlocks, they are irrelevant because the comparison is valid even if deadlocks are rare.

6. Conclusions

Choice of granularity necessarily depends on the parameters of the system. Particular values of these parameters define a window on the general granularity curve, and the choice depends on the view. The curve first decreases, then increases, and finally decreases again. Hence, *refinement of granularity* does not necessarily improve performance. The initial decrease in throughput occurs because refinement increases the transaction length, which has a greater effect on the workload than granularity. (Indeed, one's notion of the coarseness of granularity may be formalized in terms of whether refinement increases concurrency --- see [T].) The eventual decrease in throughput is a result of the interaction between the data and resource contention.

Acknowledgement We thank Nat Goodman, whose intuition and advice guided the research. We also thank Mike Carey for pointing out the importance of resource contention, and the referees for their comments and suggestions. This work was done with extensive use of MACSYMA, which is a large symbolic manipulation program developed at the MIT Laboratory of Computer Science and supported by the National Aeronautics and Space Administration under grant NSG 1323, by the Office of Naval Research under grant N00014-77-C-0641, by the U.S. Department of Energy under grant ET-78-C-4687, and by the U.S. Air Force under grant F49620-79-C-020.



N_i is the number of transactions with i locks that are executing

W_i is the number of transactions with i locks that are waiting

a_i is the restart rate of transactions holding i locks

b_i is the blocking rate of transactions holding i locks

t is the throughput of the system.

Fig. 2.1 Flow diagram for the model

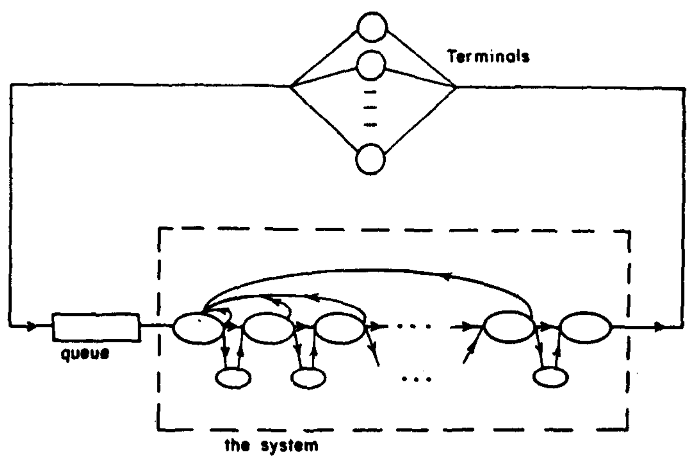
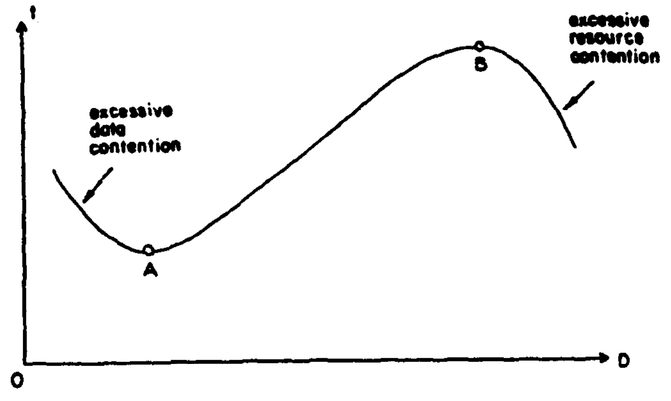
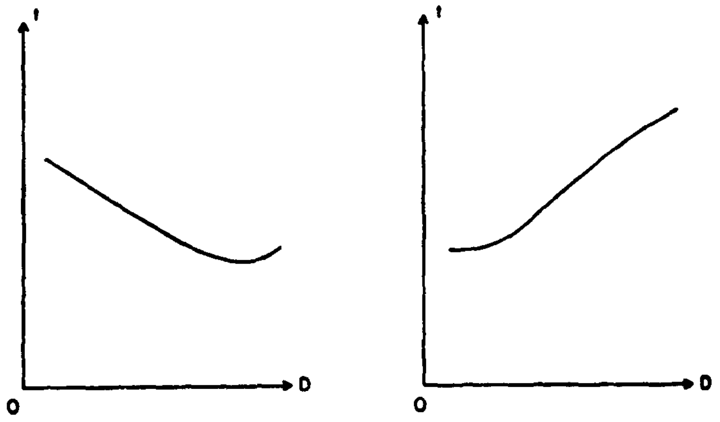


Fig. 2.2 How the flow diagram fits into a larger model for interactive systems.



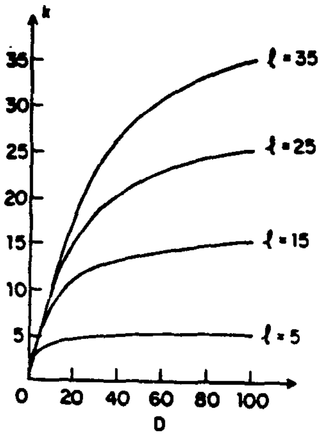
(a) The granularity curve



(b) Long transactions

(c) Short transactions

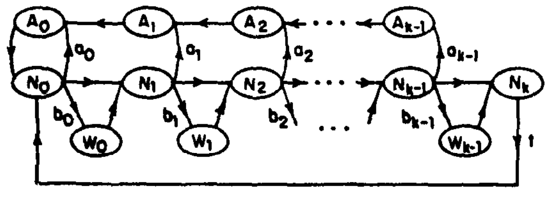
Graph 3.2



Graph 3.1

Yao's formula (3.1) for transaction length as a function of granularity

M = 100



A_i is the number of restarting transactions that are still holding i locks.

Fig. 4.1 Adding time delays for restarts to the model.

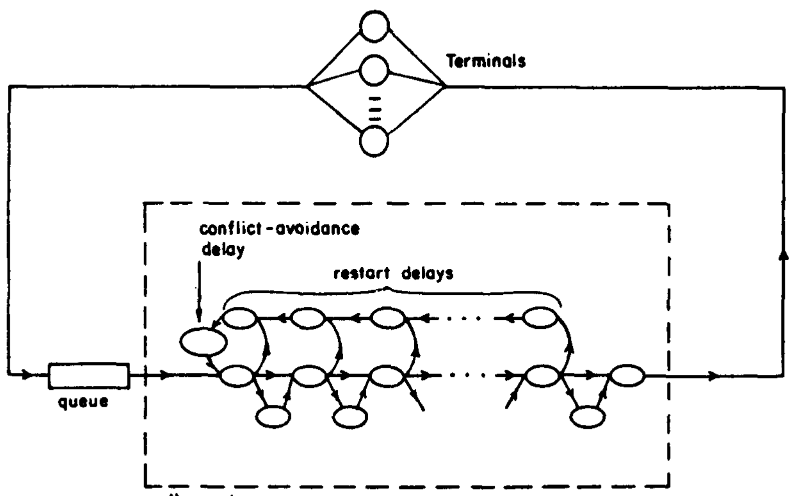
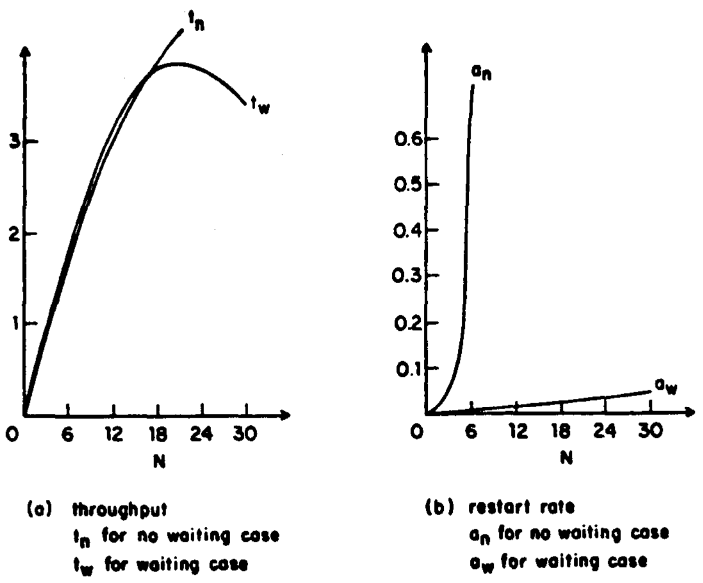
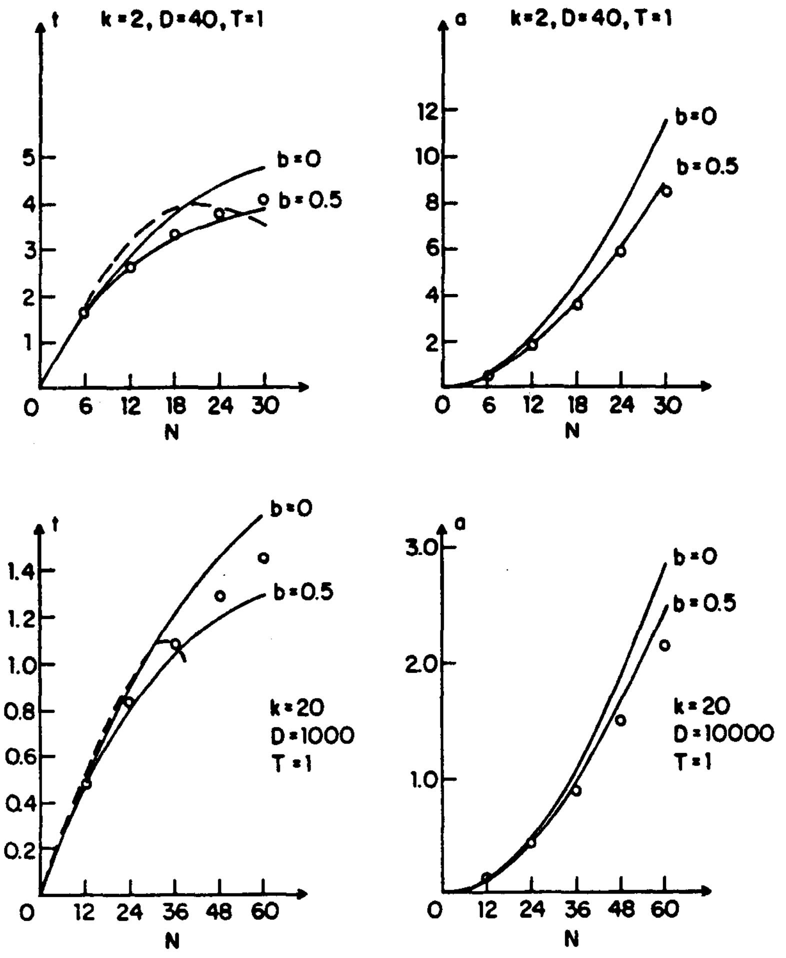


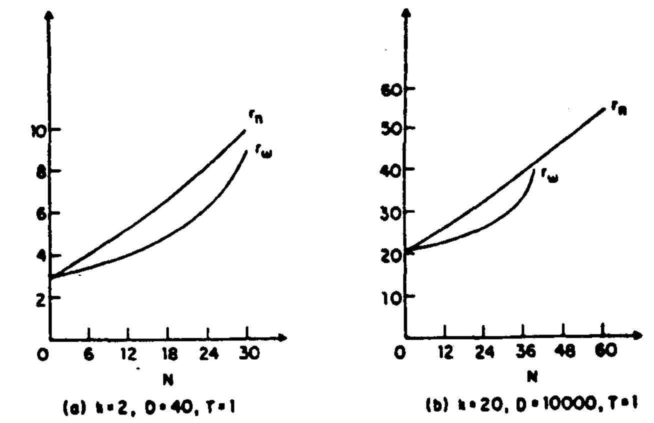
Fig. 4.2 How Fig. 2.2 is changed if conflict-avoidance and restart delays are added.



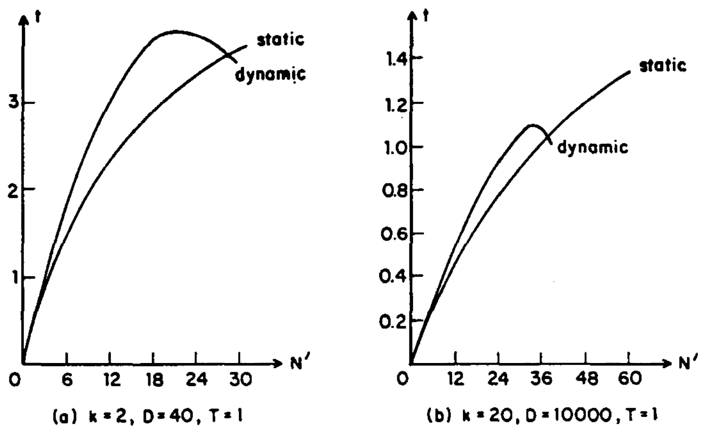
Graph 4.1 Comparison of the two cases for k=2, D=40, T=1



Graph 4.2 The effect of time delays for restarts
——— no waiting case predictions
o simulation results for b = 0.5
– – – – waiting case prediction



Graph 4.3 Comparison of response times
r_n for no waiting case
r_w for waiting case



Graph 5.1 Comparison of throughputs for dynamic and (atomic) static locking

References

[BBD] Balter, R., Berard, P., and Decitre, P. Why control of concurrency level in distributed systems is more fundamental than deadlock management. *Proc. 1st ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Ottawa, Canada (Aug. 1982), 183-193.

[BO] Beeri, C., and Obermarck, R. A resource class independent deadlock detection algorithm. *Proc International Conference on Very Large Databases*, Cannes, France (Sept. 1981), 166-178.

[C] Carey, M. Modeling and evaluation of database concurrency control algorithms. UCB/ERL 83/56, PhD dissertation, University of California, Berkeley (Sept. 1983).

[CS] Chandy, K.M., and Sauer, C.H. Approximate methods for analyzing queueing network models of computer systems. *ACM Computing Surveys 10*, 3 (Sept 1978), 281-318.

[D] Dantas, J.E.R. Performance analysis of distributed database systems. Ph.D. dissertation, Computer Science Department, University of California, Los Angeles (1980).

[DC] Devor, C., and Carlson, C.R. Structural locking mechanisms and their effect on database management system performance. *Information Systems 7*, 4 (1982), 345-358.

[DKLPS] Denning, P.J., Kahn, K.C., Leroudier, J., Potier, D., and Suri, R. Optimal multiprogramming. *Acta Informatica 7*, 2 (1976), 197-216.

[G1] Gray, J. Notes on data base operating systems. *Operating Systems -- An Advanced Course*, R. Bayer, R.M. Graham, G. Seegmuller (eds.), Springer Verlag (1978), 393-481.

[G2] Gray, J. A transaction model. *Lecture Notes in Computer Science 85*, G. Goos and J. Hartmanis (eds.), Springer Verlag (1980), 282-298.

[LN] Lin, W.K., and Nolte, J. Performance of two phase locking. *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (Feb. 1982) 131-160.

[MK] Munz, R. and Krenz, G. Concurrency in database systems — a simulation study. *Proc. ACM SIGMOD International Conference on Management of Data*, Toronto, Canada (Aug. 1977), 111-120.

[R] Ries, D.R. The effects of concurrency control on database management system performance. UCB/ERL M79/20, Ph.D. dissertation, Univ. of California, Berkeley (Apr. 1979).

[T] Tay, Y.C. A mean value performance model for locking in databases. PhD dissertation, Harvard University (Feb. 1984).

[TGS] Tay, Y.C., Goodman, N. and Suri, R. Performance evaluation of locking in databases: a survey. Manuscript in preparation.

[Y] Yao, S.B. Approximating block accesses in database organization. *Comm. ACM 20*, 4 (Apr. 1977), 260-261.